Primeiro Dia - Introdução e Sintaxe

Pedro Cavalcante

20 de fevereiro de 2019

O que é R

R é uma linguagem de programação voltada para estatística e análise de dados *open-source*, gratuita e de natureza colaborativa. Como vamos ver, o ambiente voltado para análise de dados facilita enormemente as tarefas do ciclo.

Instalar o R é muito simples, ele está disponível em http://cran.r-project.org/. É importante depois instalar o RStudio, disponível em https://www.rstudio.org/, que é uma IDE (Integrated Development Enviroment). IDEs são progrmas que facilitam, e muito, programar porque trazem um ambiente gráfico mais intuitivo, disponibilizando informações como quais objetos estão carregados na memória do computador, visualização de gráficos e animações que fazemos e por aí vai.

Links importantes

- Github

É como uma "rede social de códigos" com várias funcionalidades úteis para cuidar dos seus códigos e gerenciar projetos. É muito importante fazer uma conta lá e usar o programa para lidar adequadamente com o armazenamento dos códigos. O Github é particularmente útil para trabalhar com outras pessoas porque armazena versões antigas, quem fez que alterações nos códigos, evita conflito entre versões de programas e mantém tudo numa fonte única e facilmente acessível.

- StackOverflow

Um fórum extremamente popular em inglês sobre programação. A maior parte das suas dúvidas já foi resolvida lá e se não foi, é muito simples fazer uma nova pergunta.

- CrossValidated

Uma espécie de StackOverflow, mas voltada para análise de dados. Quando a dúvida for mais estatístisca do que de programação em si, é melhor conferir aqui. A maioria dos usuários sabe R e vai pedir algum pedaço de código para entender o seu problema bem.

- R: Uma Introdução para Economistas

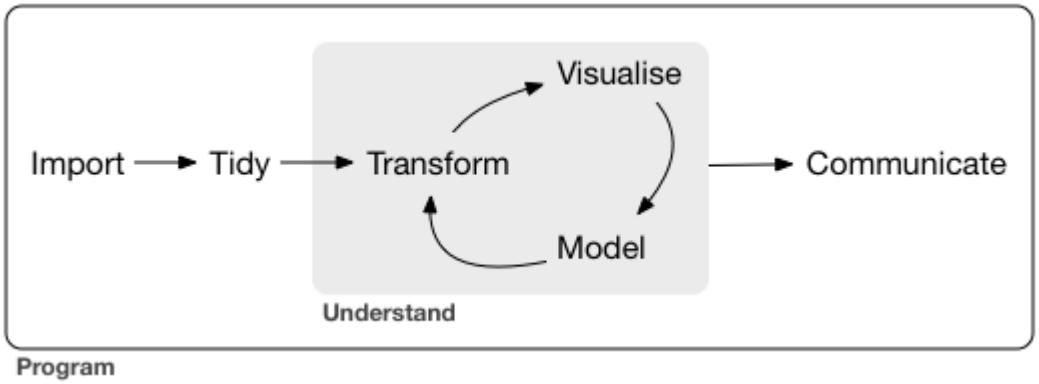

Figure 1: A caption

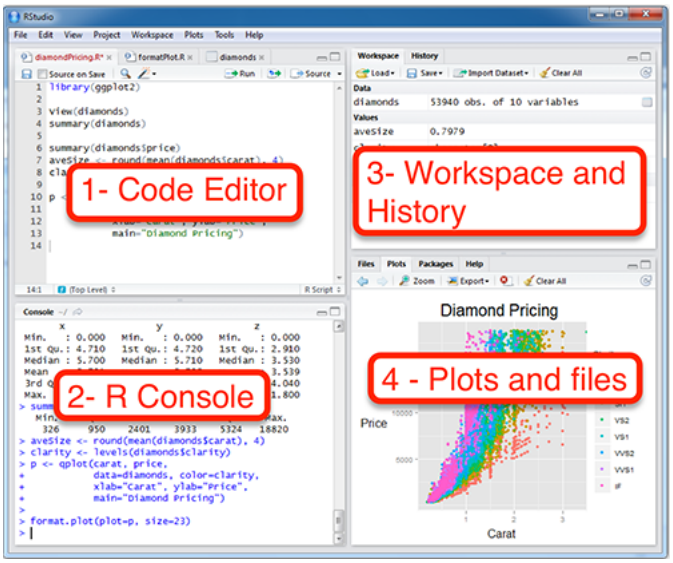

Figure 2: A caption

Uma fonte ótima para consultas rápidas.

- R for Data Science

A fonte definitiva do R introdutório. É um livro em inglês muito extenso e deve cobrir razoavelmente qualquer assunto que um iniciante queira entender melhor.

Os primeiros comandos

Tenha em mente que você pode anotar linhas de código na área do script e rodar linhas específicas copiando-as no console, digiando-as diretamente lá ou selecionando o trecho do script desejado e apertando `ctrl + enter`.

A grande vantagem do R é sua natureza colaborativa. Pesquisadores, programadores, profissionais e entusiastas do mundo todo escrevem *pacotes* com funcionalidades novas, que incluem coisas como ferramentas para econometria bayseana, gerar animaçãoes, estimar dinâmicas evolutivas, resolver problemas de otimização e gerir blogs. Pacotes tem nomes e eles trazem *funções* novas. Normalmente nos referimos a uma função na forma `pacote::função` ou `função()`. Então se lemos `dplyr::filter` sabemos que é a função `filter()` do pacote `dplyr`.

Alguns pacotes já vêm carregados no R, eles são o que chamamos de Biblioteca Padrão, a versão mais simples do R. Os mais importantes pacotes da BP são o `base` com toda a sintaxe básica da linguagem, o `stats` com dezenas de ferramentas estatísticas muito úteis e o `utils` com várias funções miscelâneas. No entanto, a maioria dos pacotes não vem carregada no R diretamente. Eles estão sediados no que chamamos de Comprehensive R Archive Network (CRAN). Podemos instalar esses pacotes facilmente e depois é simples carrega-los. Vamos baixar e carregar o `rootSolve`, que traz funções para resolver equações e problemas de cálculo diferencial.

```
install.packages("rootSolve")
library(rootSolve)
```

`install.packages()` só precisa que você dê o nome do pacote que o R instala para você. `library()` serve para carregar o pacote e ter essas funções novas disponíveis. Para saber quem fez o pacote e como devemos cita-lo em trabalhos acadêmicos, basta usar `citation()`.

```
citation("rootSolve")
```

```
##
## To cite package 'rootSolve' in publications use:
##
##   Soetaert K. and P.M.J. Herman (2009).  A Practical Guide to
##   Ecological Modelling. Using R as a Simulation Platform.
##   Springer, 372 pp.
##
##   Soetaert K. (2009).  rootSolve: Nonlinear root finding,
##   equilibrium and steady-state analysis of ordinary differential
##   equations.  R-package version 1.6
##
## rootSolve was created to solve the examples from chapter 7 of our
## book - please cite this book when using it, thank you!
## To see these entries in BibTeX format, use 'print(<citation>,
## bibtex=TRUE)', 'toBibtex(.)', or set
## 'options(citation.bibtex.max=999)'.
```

Agora vamos cobrir alguns aspectos básicos da sintaxe do R.

Uma calculadora potente

A maneira mais simples de pensar no R é como uma calculadora. Observe que podemos usar `#` para fazer comentários no código, isso é útil para deixar tudo mais legível.

```
2 + 2 # soma simples
```

```
## [1] 4
```

```
2 - 1 # subtração
```

```
## [1] 1
```

```
2^3 # elevar ao cubo
```

```
## [1] 8
```

```
2*3 #multiplicação
```

```
## [1] 6
```

```
-2*3 # multiplicação por um negativo
```

```
## [1] -6
```

```
2**3 # forma alternativa de elevar à potências
```

```
## [1] 8
```

Também podemos fazer testes lógicos usando o `==` para igual ou `!=` para diferente.

```
2 + 2 == 4 # testando se 2 + 2 = 4
```

```
## [1] TRUE
```

```
2 + 2 != 4 # agora se é diferente
```

```
## [1] FALSE
```

```r
2 + 2 > 3 # maior
```

```
## [1] TRUE
```

```r
2 + 2 < 3 # menor
```

```
## [1] FALSE
```

```r
TRUE == FALSE # podemos também testar proposições lógicas mais abstratas
```

```
## [1] FALSE
```

No entanto, a maior parte do tempo lidaremos com *objetos*, que iremos definir com o sinal `<-`, que chamamos de Operador de Designação (Assignment Operator). Se parecer muito estranho digitar isso, o atalho é `Alt` + `-`. Você também pode usar o sinal `=`, ele vai funcionar quase sempre como um sinônimo, mas talvez possa te confundir se estiver fazendo testes lógicos, então use com cuidado.

```r
a = 2 # definindo um objeto a como 2
b <- 2 # o mesmo com b, usando o sinal <-

a == b # teste lógico
```

```
## [1] TRUE
```

Temos também como usar funções, pedaços prontos de código com funcionalidades específicas. Funções podem ou não admitir *argumentos*, que são especificados usando seu nome, um sinal de `=` e o valor do argumento, todos separados por vírgula. Por questões de organização de código, é bom pular uma linha para cada argumento, embora você possa usar formas alternativas de identação do texto. Algumas funções são simples o suficiente para que você não precise dizer exatamente qual argumento é qual, como é o caso de `seq()`. `seq()` também tem outra pecualiaridade, seu argumento `by`, que informa o tamanho do passo entre um elemento e outro da sequência é por padrão o número 1. Descobrimos isso lendo a documentação da função. Você pode acessa-la pela função `help()` ou apertando `F1` quando o cursor estiver em cima da funçaõ.

```r
help(seq)
```

Abaixo listo algumas funções com funcionalidades muito básicas como `print()` que retorna no console o valor de algum objeto, `exp(x)` que calcula e^x e `seq()` para gerar sequências.

```r
print(a) # retorna no console o valor de a
```

```
## [1] 2
```

```r
exp(4) # exponencial
```

```
## [1] 54.59815
```

```r
factorial(4) # fatorial
```

```
## [1] 24
```

```r
sqrt(9) # raiz quadrada
```

```
## [1] 3
```

```r
choose(4, 2) # permutação de 4, 2 a 2
```

```
## [1] 6
```

```r
seq(from = 1,
    to = 10,
    by = 2) #sequência de-para com passo 2
```

```
## [1] 1 3 5 7 9
```

```r
seq(1, 10, 2) # o mesmo resultado sem especificar qual argumento é qual
```

```
## [1] 1 3 5 7 9
```

```r
c = seq(1, 5, 1) # agora com um passo 1

1:5 # usar : também serve para gerar sequências com passo 1
```

```
## [1] 1 2 3 4 5
```

```r
print(c)
```

```
## [1] 1 2 3 4 5
```

```r
sum(c) # soma das entradas
```

```
## [1] 15
```

```r
mean(c) # média de c
```

```
## [1] 3
```

Observe que o objeto `c` é um pouco diferente dos anteriores, que eram só um número. `c` tem uma sequência. Para descobrir a classe de um objeto, usamos a função `class()` e para inspeciona-lo melhor usar `str()` (uma abreviação de *estrutura* em inglês).

```r
class(a)
```

```
## [1] "numeric"
```

```r
str(a)
```

```
##  num 2
```

```r
class(c)
```

```
## [1] "numeric"
```

```r
str(c)
```

```
##  num [1:5] 1 2 3 4 5
```

Uma das estruturas de dados mais importantes do R são *vetores*. Podemos declarar vetores de várias formas, uma "simples" é usando a função `c()`. Para saber se um objeto é vetor, podemos usar `is.vector()`. Objetos podem ser nomeados com números, desde que não comecem com um número. Letras maísculas e minúsculas são diferenciadas, então tenha isso em mente ao nomear objetos.

```r
A = c(2,2) # A é um vetor de duas dimensões em que cada entrada é um 2

print(A) # printamos no console
```

```
## [1] 2 2
```

```r
class(A) # descobrimos a classe
```

```
## [1] "numeric"
```

```r
str(A) # inspecionamos a estrutura
```

```
##  num [1:2] 2 2
```

```r
is.vector(A)
```

```
## [1] TRUE
```

Vetores são estruturas de dados muito flexíveis, podemos armazenas de tudo neles. Um truque para se poupar de escrever muitos prints se precisar é escrever a linha de código toda entre parênteses. Como por exemplo:

```
(B = c(2, -3, 5, 8))
```

```
## [1]  2 -3  5  8
```

```
C = c("um", "dois", "madeira", "peixe", "PET-UFF", "Niterói")

D = c(TRUE, FALSE, TRUE, FALSE, FALSE)
```

Vale parar brevemente para falar de fatores. Até agora trabalhamos com dados lógicos ou numéricos, mas é muito comum encontrar dados categóricos como por exemplo sexo ou profissão. É mais simples lidar com esse tipo de dado quando é declarado como um fator. Isso é simples, basta usar a função `factor()`. Esse tipo de classe é muito útil para rodar modelos com variáveis categóricas porque o R faz por nós o trabalho de criar variáveis dummy com cada classe e nos informa quais tipos foram observados. Para lidar com fatores também é comum usar `levels()`, que retorna os tipos encontrados no vetor. Digamos por exemplo que `C2` seja uma lista de extensções de que participam 10 alunos aleatoriamente escolhidos.

```
C2 = c("PET", "Atlética", "PET", "Goal", "Opção", "PET", "Opção", "Atlética", "Opção", "PET")
C2 = factor(C2)

C2
```

```
##  [1] PET      Atlética PET      Goal     Opção    PET      Opção
##  [8] Atlética Opção    PET
## Levels: Atlética Goal Opção PET
```

```
levels(C2)
```

```
## [1] "Atlética" "Goal"     "Opção"    "PET"
```

Operações com vetores são bem intuitivas no R porque a maioria das funções é vetorizada - são aplicadas à todos os elementos de um vetor. A função `ifelse()` por exemplo - que deve lembrar aos usuários de Excell a função `SE` - também funciona no mesmo espírito. Basta especificarmos um teste lógico, uma resposta para verdadeiro e outra para falso.

```
E = c(1, 3, 4, 9)

F_ = B*E # multiplicação de vetores

# nunca declare um objeto chamado F ou T porque são os símbolos de verdadeiro e falso

B*E # podemos também somente recuperar a conta se não quisermos printar F_
```

```
## [1]  2 -9 20 72
```

```
G = ifelse(C == "PET-UFF", # testa se cada entrada é igual a "PET-UFF"
           "Entrada do PET", # resposta se for
           "Não é a Entrada do PET") # resposta se não for
print(G)
```

```
## [1] "Não é a Entrada do PET" "Não é a Entrada do PET"
## [3] "Não é a Entrada do PET" "Não é a Entrada do PET"
## [5] "Entrada do PET"         "Não é a Entrada do PET"
```

O próximo objeto são matrizes. Matrizes precisam de vetores do mesmo tipo para funcionar. Precisamos alimentar um vetor só e depois especificar quantas linhas e colunas queremos. Podemos pedir os autovalores e autovetores da matriz e também podemos multiplicar matrizes com `%*%`.

```r
H = c(1, 3, 2, 4)

I = matrix(H,        # vetor
           nrow = 2, # número de linhas
           ncol = 2) # número de colunas

auto = eigen(I) #autovalores e autovetores da matriz
print(auto)
```

```
## eigen() decomposition
## $values
## [1]  5.3722813 -0.3722813
##
## $vectors
##            [,1]       [,2]
## [1,] -0.4159736 -0.8245648
## [2,] -0.9093767  0.5657675
```

```r
J = c(2, 1, 5, 3)

K = matrix(J,
           ncol = 2)

I %*% K ## multiplicação de matrizes
```

```
##      [,1] [,2]
## [1,]    4   11
## [2,]   10   27
```

Exercícios

- Calcule $52e^2 - 35 \times 4$!
- Ache, se existirem, os autovetores da matriz:

$$A = \begin{bmatrix} 1 & 2 & 5 \\ 0 & 3 & 1 \\ 2 & 4 & 3 \end{bmatrix}$$

- Ache a transposta de A
- Calcule a soma dos autovalores da matriz A.
- Calcule a média de uma sequência começando em 150, terminando em 500, com passo $0,5$

Data Frames

A estrutura de dados mais comum é um Data Frame. Usamos a função `data.frame()` para gera-los. Um DF é uma coleção de vetores que admite tipos *diferentes*, então são mais flexíveis que matrizes. Na hora de declarar o DF, podemos dar nome a cada vetor. Observe que precisamos que todos os vetores do DF tenham o *mesmo* comprimento. Podemos usar a função `length()` para averiguar isso.

```r
elemento1 = seq(1, 100)
elemento2 = seq(50, 150)

length(elemento1)
```

```
## [1] 100
```

```r
length(elemento2)
```

```
## [1] 101
```

```r
base = data.frame(primeiro = elemento1,
                  segundo = elemento2)
```

```
## Error in data.frame(primeiro = elemento1, segundo = elemento2): arguments imply differing number of r
```

No entanto se fizermos `elemento1` e `elemento2` terem o mesmo comprimento, o DF sai sem problemas. A função `head()` puxa o cabeçalho de um objeto e é muito útil para averiguar visualmente se está tudo como esperado.

```r
elemento1 = seq(1, 100)
elemento2 = seq(50, 149)

length(elemento1)
```

```
## [1] 100
```

```r
length(elemento2)
```

```
## [1] 100
```

```r
base = data.frame(primeiro = elemento1,
                  segundo = elemento2)

head(base)
```

```
##   primeiro segundo
## 1        1      50
## 2        2      51
## 3        3      52
## 4        4      53
## 5        5      54
## 6        6      55
```

Nos referimos aos elementos de um DF pelo símbolo `$`. Então se quisermos resgatar somente o vetor `primeiro`, usamos `base$primeiro`. Isso também vale se quisermos criar mais vetores na base. Também podemos nos referir a elementos específicos usando chaves.

```r
base$terceiro = base$primeiro + base$segundo

mean(base$terceiro) # média
```

```
## [1] 150
```

```r
median(base$primeiro) # mediana
```

```
## [1] 50.5
```

```r
summary(base$segundo) # sumário estatístico
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   50.00   74.75   99.50   99.50  124.25  149.00
```

```r
rowMeans(base) # média de cada linha
```

```
##   [1]  34.00000  35.33333  36.66667  38.00000  39.33333  40.66667  42.00000
##   [8]  43.33333  44.66667  46.00000  47.33333  48.66667  50.00000  51.33333
##  [15]  52.66667  54.00000  55.33333  56.66667  58.00000  59.33333  60.66667
```

```
##  [22]  62.00000  63.33333  64.66667  66.00000  67.33333  68.66667  70.00000
##  [29]  71.33333  72.66667  74.00000  75.33333  76.66667  78.00000  79.33333
##  [36]  80.66667  82.00000  83.33333  84.66667  86.00000  87.33333  88.66667
##  [43]  90.00000  91.33333  92.66667  94.00000  95.33333  96.66667  98.00000
##  [50]  99.33333 100.66667 102.00000 103.33333 104.66667 106.00000 107.33333
##  [57] 108.66667 110.00000 111.33333 112.66667 114.00000 115.33333 116.66667
##  [64] 118.00000 119.33333 120.66667 122.00000 123.33333 124.66667 126.00000
##  [71] 127.33333 128.66667 130.00000 131.33333 132.66667 134.00000 135.33333
##  [78] 136.66667 138.00000 139.33333 140.66667 142.00000 143.33333 144.66667
##  [85] 146.00000 147.33333 148.66667 150.00000 151.33333 152.66667 154.00000
##  [92] 155.33333 156.66667 158.00000 159.33333 160.66667 162.00000 163.33333
##  [99] 164.66667 166.00000
```

```
colMeans(base) # média de cada coluna
```

```
## primeiro  segundo terceiro
##     50.5     99.5    150.0
```

```
base[1,2] # pega o elemento na primeira linha e segunda coluna
```

```
## [1] 50
```

```
base[1,] # pega a primeira linha
```

```
##   primeiro segundo terceiro
## 1        1      50       51
```

```
base[base$terceiro > 30,] # pega só as linhas em que a variável terceiro é maior que 30, vale para outr
```

```
##     primeiro segundo terceiro
## 1          1      50       51
## 2          2      51       53
## 3          3      52       55
## 4          4      53       57
## 5          5      54       59
## 6          6      55       61
## 7          7      56       63
## 8          8      57       65
## 9          9      58       67
## 10        10      59       69
## 11        11      60       71
## 12        12      61       73
## 13        13      62       75
## 14        14      63       77
## 15        15      64       79
## 16        16      65       81
## 17        17      66       83
## 18        18      67       85
## 19        19      68       87
## 20        20      69       89
## 21        21      70       91
## 22        22      71       93
## 23        23      72       95
## 24        24      73       97
## 25        25      74       99
## 26        26      75      101
## 27        27      76      103
```

```
## 28        28       77       105
## 29        29       78       107
## 30        30       79       109
## 31        31       80       111
## 32        32       81       113
## 33        33       82       115
## 34        34       83       117
## 35        35       84       119
## 36        36       85       121
## 37        37       86       123
## 38        38       87       125
## 39        39       88       127
## 40        40       89       129
## 41        41       90       131
## 42        42       91       133
## 43        43       92       135
## 44        44       93       137
## 45        45       94       139
## 46        46       95       141
## 47        47       96       143
## 48        48       97       145
## 49        49       98       147
## 50        50       99       149
## 51        51      100       151
## 52        52      101       153
## 53        53      102       155
## 54        54      103       157
## 55        55      104       159
## 56        56      105       161
## 57        57      106       163
## 58        58      107       165
## 59        59      108       167
## 60        60      109       169
## 61        61      110       171
## 62        62      111       173
## 63        63      112       175
## 64        64      113       177
## 65        65      114       179
## 66        66      115       181
## 67        67      116       183
## 68        68      117       185
## 69        69      118       187
## 70        70      119       189
## 71        71      120       191
## 72        72      121       193
## 73        73      122       195
## 74        74      123       197
## 75        75      124       199
## 76        76      125       201
## 77        77      126       203
## 78        78      127       205
## 79        79      128       207
## 80        80      129       209
## 81        81      130       211
```

```
## 82        82     131      213
## 83        83     132      215
## 84        84     133      217
## 85        85     134      219
## 86        86     135      221
## 87        87     136      223
## 88        88     137      225
## 89        89     138      227
## 90        90     139      229
## 91        91     140      231
## 92        92     141      233
## 93        93     142      235
## 94        94     143      237
## 95        95     144      239
## 96        96     145      241
## 97        97     146      243
## 98        98     147      245
## 99        99     148      247
## 100      100     149      249
```

E também temos listas. São formas bem gerais de estruturas de dados, porque adimitem qualquer coisa. O primeiro elemento de uma lista pode ser um DF, o segundo um vetor e o terceiro uma letra.

```
lista = list(primeiro = base,
             segundo = seq(1, 10),
             terceiro = "a")

class(lista)
```

```
## [1] "list"
```

```
str(lista)
```

```
## List of 3
##  $ primeiro:'data.frame':    100 obs. of  3 variables:
##   ..$ primeiro: int [1:100] 1 2 3 4 5 6 7 8 9 10 ...
##   ..$ segundo : int [1:100] 50 51 52 53 54 55 56 57 58 59 ...
##   ..$ terceiro: int [1:100] 51 53 55 57 59 61 63 65 67 69 ...
##  $ segundo : int [1:10] 1 2 3 4 5 6 7 8 9 10
##  $ terceiro: chr "a"
```

```
lista$terceiro
```

```
## [1] "a"
```

Lembra quando tiramos os autovalores de uma matriz? Salvamos eles no objeto `auto`. Bem, como várias funções, `eigen()` retorna uma lista com elementos. Em particular, `eigen()` retorna um tipo particular de lista chamado *eigen* em que a primeira entrada é um vetor com os autovalores e a segunda é uma matriz com os autovetores.

```
auto$values
```

```
## [1]  5.3722813 -0.3722813
```

```
auto$vectors
```

```
##            [,1]       [,2]
## [1,] -0.4159736 -0.8245648
## [2,] -0.9093767  0.5657675
```

Antes de prosseguir para ambientes controlados, vamos parar para falar brevemente de Pastas de Trabalho. Elas são importantes porque vão facilitar demais a sua vida. Sempre que você precisar que o R leia um arquivo fora da pasta de trabalho, vai precisar dar o endereço *completo* dele, o que é chato, apesar de simples. Para definir um endereço de trabalho, é só coloca-lo entre aspas na função `setwd()`. Para descobrir qual é o endereço atual, basta usar `getwd()` sem argumentos.

Ambientes controlados

Ambientes controlados são maneiras de organizar testes lógicos e ações a serem tomadas para resultados diferentes. Vamos cobrir as duas funções mais comuns, o loop `for()` e o ambiente `if()` e expandir um pouco nossa capacidade de fazer testes lógicos com os operadores E e OU.

`2+2 == 4` sempre irá retornar um verdadeiro, mas se testarmos se esse enunciado é verdadeiro conjuntamente com outro as garantias vão embora. Para juntar enunciados lógicos no R usando o conectivo E usamos a letra `&`.

```
2 + 2 == 4
```

```
## [1] TRUE
```

```
2 + 2 == 4 & 3 + 3 == 6 # uma verdadeira & uma verdadeira
```

```
## [1] TRUE
```

```
2 + 2 == 4 & 3 - 2 == -1 # uma verdadeira & uma falsa
```

```
## [1] FALSE
```

Também podemos testar se uma ou outra são verdadeiras, nesse caso usamos a barra vertical `|`, acionada com `Shift + \`.

```
2 + 2 == 4 | 3 + 3 == 6 # uma verdadeira ou uma verdadeira
```

```
## [1] TRUE
```

```
2 + 2 == 4 | 3 - 2 == -1 # uma verdadeira ou uma falsa
```

```
## [1] TRUE
```

O enunciado `if`

O enunciado `if` segue sempre a mesma estrutura:

```
if(Condição == Verdadeira) {

  Expressão

  }
```

Digamos que temos um vetor com dados de vendas mensais e uma meta, poderíamos então fazer:

```
meta = 200
vendas_mensais = c(12, 15, 18, 25, 30, 16, 20, 12, 13, 15, 16, 22)

if(sum(vendas_mensais) > meta) {

  print("Meta de vendas cumprida")

  }
```

```
## [1] "Meta de vendas cumprida"
```

Podemos rebuscar um pouco isso usando funções como `paste()` que agrupa pedaços de texto e `else` para definir o que deve ser feito caso o teste lógico retorne Falso.

```
if(sum(vendas_mensais) > meta) {

  diferenca = sum(vendas_mensais) - meta

  print(paste("Meta de vendas cumprida com margem de", diferenca))

  }
```

```
## [1] "Meta de vendas cumprida com margem de 14"
```

Agora com um `else` a estrutura é essencialmente a mesma:

```
if(Condição == Verdadeira) {

  Expressão

} else {

  Expressão alternativa

  }
```

Como por exemplo:

```
if(sum(vendas_mensais) > meta) {

  diferenca = sum(vendas_mensais) - meta

  print(paste("Meta de vendas cumprida com margem de", diferenca))

} else {

  diferenca = sum(vendas_mensais) - meta

  print(paste("Meta de vendas não foi cumprida, com diferença de", diferenca))

  }
```

```
## [1] "Meta de vendas cumprida com margem de 14"
```

Observe que repetimos o cálculo da diferença dentro de cada opção. Isso é importante porque o que quer que esteja dentro das chaves só vai ser executado se o teste lógico retornar um resultado específico. Poderíamos definir também a diferença do lado de fora do `if()` para não precisarmos repetir.

```
diferenca = sum(vendas_mensais) - meta

if(sum(vendas_mensais) > meta) {

  print(paste("Meta de vendas cumprida com margem de", diferenca))

} else {

  print(paste("Meta de vendas não foi cumprida, com diferença de", diferenca))

  }
```

```
## [1] "Meta de vendas cumprida com margem de 14"
```

O loop `for()`

Existem outras formas de loop, mas vamos por enquanto focar no mais útil, o loop for. Sempre usaremos loops for quando precisarmos realizar operações elemento por elemento.

Genericamente, um loop for tem a forma:

```
for (i in Lista de índices) {

  Expressão(i)

  }
```

Se temos uma sequência numérica e queremos saber a soma acumulada até o i-ésimo elemento, basta montar um loop. Se der tudo certo o último elemento do vetor `soma_acumulada` será 5050.

```
numeros = seq(1, 100)
soma_acumulada = vector() # declaramos um vetor vazio

for(i in 1:100) {

  soma_acumulada[i] = sum(numeros[1:i]) # preenchemos o vetor vazio
  #numeros[1:i] pega todos os elementos de "numeros" entre o primeiro e o i-ésimo
}

print(soma_acumulada)
```

```
##   [1]    1    3    6   10   15   21   28   36   45   55   66   78   91  105
##  [15]  120  136  153  171  190  210  231  253  276  300  325  351  378  406
##  [29]  435  465  496  528  561  595  630  666  703  741  780  820  861  903
##  [43]  946  990 1035 1081 1128 1176 1225 1275 1326 1378 1431 1485 1540 1596
##  [57] 1653 1711 1770 1830 1891 1953 2016 2080 2145 2211 2278 2346 2415 2485
##  [71] 2556 2628 2701 2775 2850 2926 3003 3081 3160 3240 3321 3403 3486 3570
##  [85] 3655 3741 3828 3916 4005 4095 4186 4278 4371 4465 4560 4656 4753 4851
##  [99] 4950 5050
```

Observe que vários parâmetros do loop podem se adaptar automaticamente aos dados com um pouco de imaginação. Vamos acessar dados prontos com a função `data()`. Mais especificamente a base iris, com dados de algumas espécies de flores. A função `head()` mostra as primeiras linhas da base para que tenhamos uma ideia do que ela mostra e como está estruturada.

```
data(iris)

head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

Digamos que não saibamos quantas variáveis a base tem e queremos fazer um loop que diga a classe de todos os vetores. Lembre-se que para acessar elementos específicos de DataFrames e Listas não podemos usar só uma chave `[]`, precisamos usar duas `[[]]`

```r
for(i in 1:ncol(iris)) { #ncol() pega o número de colunas de um dataframe

  print(class(iris[[i]]))

}
```

```
## [1] "numeric"
## [1] "numeric"
## [1] "numeric"
## [1] "numeric"
## [1] "factor"
```

Exercícios

- Gere um DataFrame que é uma "grade" 100x100. Você terá então um DataFrame com 100 x 100 linhas e duas colunas, cada uma representando uma coordenada de um espaço de duas dimensões. Mais explicitamente, a primeira coluna terá cem vezes o número 1, cem vezes o número 2 e assim em diante, com a segunda tendo cem vezes a sequência de 1 a 100.
- Faça um loop para obter a média de todas as variáveis da base de dados `longley`
- A base `LifeCycleSavings` contém dados de estrutura etária e renda. Localize a documentação da base para descobrir o que é cada variável e defina um DataFrame que contém todos os países com taxa de crescimento da renda disponível maior do que 3%.
- A base `mtcars` contém dados de modelos diferentes de carros e todas as suas variáveis são numéricas. Transforme todas as variáveis categórias, o número de cilindros, de marchas e de carburadores em fatores.

(Extra) Criação de funções

Grande parte da rotina de analisar dados é um tanto quanto repetitiva. Observe que comandos muito frequentes como loops e gerar sequências têm funções prontas. Sempre que encararmos uma atividade repetida, é uma boa ideia criar uma função nova automizando seu procedimento repetido usando `function()`. Declaramos um objeto com um nome e a ele passamos a chamada dessa função primordial. Os argumentos de `function()` são justamente os argumentos da função que você está criando, e com certa flexibilidade. Algumas funções muito simples sequer precisam de argumentos, como `getwd()` por exemplo.

```r
mensagem = function(){
  print("Insira aqui uma frase motivacional")
}

#evocando a função
mensagem()
```

```
## [1] "Insira aqui uma frase motivacional"
```

Podemos fazer funções que dependam de argumentos a serem totalmente especificados. Como normalmente isso envolve algum tipo de manipulação e solução de algoritimo, queremos devolver ao usuário a solução disso. Usamos a função `return()` para isso.

```r
soma.minha = function(x, y){
  valor = x + y
  return(valor)
}

soma.minha(2,4)
```

```
## [1] 6
```

```
soma.minha(2,"h")
```

```
## Error in x + y: non-numeric argument to binary operator
```

Funções, assim como código normal, aceitam controle de fluxo e isso é muito útil para lidar com possíveis erros cometidos pelo usuário. Existem alguns pormenores aqui como por exemplo as funções `stop()`, `message()`, `warning()` e `print()` ou sutilezas como a diferença entre `library()` e `require()` ou mesmo `require.Namespace()`. Caso vá escrever funções que serão usadas por mais gente, é importante se informar sobre essas detalhes mais finos. Uma boa fonte é o livro Advanced R de Hadley Wickham.

```
soma.minha = function(x,y){
  if(!is.numeric(x) | !is.numeric(y)){
    print("Você deve inserir dois números para que a função possa ser executada")
  }else{
    valor = x + y
    return(valor)
  }
}

soma.minha(2,"h")
```

```
## [1] "Você deve inserir dois números para que a função possa ser executada"
```

Alguns argumentos não precisam ser especificados *sempre*. Lidando com dados, estamos quase sempre interessados em alguns limiares de significância por exemplo, ou quando vamos trabalhar quase sempre com uma certa taxa de erro aceitável. Para isso especificamos valores padrão para argumentos.

```
###argumentos com padrão

potencia = function(x, base = 10){
  valor = base^x
  return(valor)
}

potencia(3)
```

```
## [1] 1000
```